by
Jeff Prosise

Tutor

HOW DO THEY RUN?

Would you describe how the three common types of mice work (serial, bus, and PS/2 mouse port), how the mouse driver communicates with the mouse, and how programs in turn talk to the mouse driver?

Jacek Hejnar
St. Hubert, Quebec
Canada

■ **HOW DO THEY RUN? A look at the three common mouse interfaces.**

At the software level, bus mice, serial mice, and PS/2 mice are all the same. The mouse driver provides a common interface so that an application program can communicate with a mouse using identical sets of function calls regardless of its type.

At the hardware level, however, these mouse types are fundamentally different. A serial mouse works just like a communications device: It sends data to the serial port of your choice each time a mouse event takes place. (A *mouse event* is when a button is pressed or released or the mouse is moved.) A microprocessor inside the mouse transmits a packet of data any time it detects a change in state.

Typically, the mouse driver programs the serial port to interrupt the CPU when a byte of data is received at the port. It does this in the same way a communications program such as *ProComm* would. If the mouse is stationary and no buttons are being activated, no CPU time is tied up tracking mouse actions.

Many serial mice use the 3-byte packet format standardized by Microsoft to transmit information about mouse position and the state of the buttons to the CPU. Figure 1 shows the format of the packet. Since the Microsoft Serial Mouse transmits data at 1,200 bits per second using 1 stop bit, 7 data bits, and no parity, bit 7 (the leftmost bit) in each byte isn't used. Bit 6 is a clock bit that the mouse driver uses to stay in sync with the mouse. In the first byte in the packet, it's set to 1; in the second and third bytes, it's set to 0. Bits 4 and 5 indicate the current up/down states of the left and right mouse buttons. A 0 indicates that the corresponding button is pressed, while a 1 indicates that it's not. The remaining bits in the data packet are combined to form two 8-bit numbers indicating the number of mickeys the mouse has moved in the *x* (horizontal) and *y* (vertical) directions since the last report. A *mickey* is the mouse unit of distance, equal to approximately 0.005 inch. By recording mickeys, the mouse driver can track the location of the mouse at all times.

The mouse driver can determine the direction of the movement—up or down, right or left—from the sign of the mickey counts transmitted by the mouse, which are encoded in two's complement form (a common method for representing signed integers on computers). Using this representation, an 8-bit value can store any number from –128 to +127, inclusive. Thus, the mouse can move up to 127 units in either direction (about 0.62 inch) in the time it takes a packet to be transmitted (about 20 milliseconds at 1,200 bps) without missing a beat. That provides enough latitude for you to move the mouse across the table rather quickly, without losing any information in the process. Some mice use higher data rates to increase resolution. Logitech's Series 9 Mouse, for example, operates at 2,400 bps, doubling the range that the mouse may travel between data transmissions.

MICROSOFT SERIAL MOUSE DATA PACKET FORMAT

Byte 1							
x	1	LB	RB	Y7	Y6	X7	X6
Byte 2							
x	0	X5	X4	X3	X2	X1	X0
Byte 3							
x	0	Y5	Y4	Y3	Y2	Y1	Y0

Bit	Description
x	Not used
1	Always 1
0	Always 0
LB	Left button 0 = pressed 1 = not pressed
RB	Right button 0 = pressed 1 = not pressed
X7–X0	8-bit *x*-movement indicator (number of mickeys moved in the horizontal direction since the last data transmission)
Y7–Y0	8-bit *y*-movement indicator (number of mickeys moved in the vertical direction since the last data transmission)

Figure 1: Each time a serial mouse is moved, it transmits a packet of data containing information about mouse position and the state of the buttons to the CPU. This figure illustrates the 3-byte data packet format that is used by the Microsoft Serial Mouse and many other Microsoft-compatible serial mice.

Not all serial mice use Microsoft's 3-byte packet format. The three-button Mouse Systems Mouse, for example, transmits data in 5-byte packets. The added bits are used to transmit information about the third button (something that the Microsoft format does not support) and information about the mouse's current and last positions, which, among other things, can be used to determine mouse velocity.

Fortunately, programs don't have to interface with the mouse at the packet level; that's what mouse drivers are for.

How does the serial mouse get its power? It's not battery operated, nor does it have to be plugged into an electrical outlet. Instead, it draws power directly from the serial port through the RTS (request to send) line. The Microsoft Serial Mouse uses five RS-232 lines: TD (transmit data), RD (read data), DTR (data terminal ready), RTS (ready to send), and SG (signal ground). TD is used to transmit packets of data. The driver asserts RTS when it is activated, to make sure power is available to the mouse, and asserts DTR as a signal that it's installed and ready to go. In current Microsoft Serial Mouse implementations, RD is unused.

THE BUS MOUSE

The bus mouse interfacing scheme takes an entirely different approach to linking a mouse to a PC. Unlike the serial mouse, the bus mouse does not contain its own microprocessor. Instead, logic on the bus interface card is responsible for monitoring the mouse and notifying the mouse driver when the mouse is moved or a button is activated. In the most common implementation, the card is programmed to poll the mouse at regular intervals (typically every 1/30 to 1/60 second) and interrupt the CPU so that the mouse driver can read the current mouse status from registers on the card. Why these interrupt rates? Because 30 to 60 Hz roughly corresponds to the refresh rates of most displays. When the mouse pointer is being moved across the screen, there's usually no need to poll the mouse more frequently, because the screen can't be updated fast enough to show what's happening.

Variations on this basic theme make it difficult to paint a picture of how a "typical" bus mouse operates. Versions of the Microsoft Bus Mouse sold since 1986, for example, contain a custom chip called an *InPort* on the bus interface card. This chip tracks mouse events and—like the microprocessor in the serial mouse—interrupts the CPU only when the mouse is moved or a mouse button is pressed or released. This way, valuable CPU time isn't wasted reading registers on the interface card when the mouse is idle. And thanks to the InPort chip, the Microsoft Bus Mouse can be programmed for 30-, 50-, 100-, or 200-Hz operation. It's also possible to have the InPort chip interrupt the CPU at regular intervals regardless of whether there's anything new to report, or to forgo interrupts altogether and allow software to drive the mouse strictly by polling it. Using this scheme, the mouse driver (or a program driving the mouse directly) can track mouse position more precisely than programmed 30-to-200-Hz interrupt rates allow. Alternatively, it can tap into the vertical sync pulse generated by the video adapter each time a screen refresh is completed and take time out to read the mouse status during the vertical blanking interval.

A bus mouse does not send packets of data the way the serial mouse does. Instead, status is read directly from lines connecting the mouse to the bus interface card. For example, the Microsoft Bus Mouse uses the nine-pin Hosiden circular connector shown in Figure 2. Of the nine pins, three—SW1, SW2, and SW3—reflect the state of up to three mouse buttons in real time. XA, XB, YA, and YB carry *quadrature* signals that communicate the amount and direction of motion in the *x* and *y* directions to counter registers inside the InPort chip. In quadrature encoding, two lines are provided to track motion in a given direction. A pulse on one of the lines—for example, XA—indicates that the mouse has been moved. A pulse on XB, which arrives slightly out of phase with the pulse on XA, reveals the direction of the movement. If XA leads, then movement is positive; if the pulse on XB comes first, movement is in the negati direction. Values read from the InPort s counter registers are two's complement 8-bit numbers, so they, like the values transmitted by the serial mouse, may range from –128 to +127.

The remaining two pins on the connector serve equally important functions. The bus mouse draws its power from the +5V pin, while the other pin goes to ground.

THE PS/2 MOUSE

The PS/2 mouse is similar to the serial mouse in many respects. This mouse contains its own microprocessor that transmits clocked serial data to the keyboard controller inside the PS/2, just as the PS/2 keyboard does. The controller decodes the information coming in and the PS/2 BIOS makes it available to the mouse driver. Through the BIOS programming interface, the mouse sampling rate may be set to frequencies ranging from 10 to 200 Hz. Typically, the mouse driver will register itself with the BIOS, and the BIOS will activate a handler inside the driver each time a mouse-related event occurs.

One peculiarity of the PS/2 mouse is

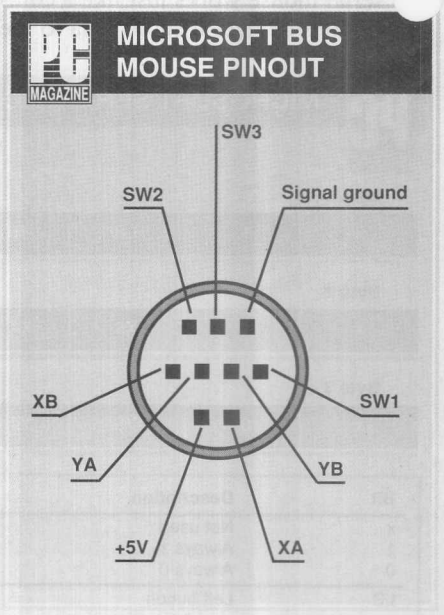

Figure 2: The Microsoft Bus Mouse uses a nine-pin Hosiden circular connector to interface with the bus interface card. SW1, SW2, and SW3 are used to sense the state of the mouse buttons, while XA, XB, YA, and YB carry quadrature signals indicating m se movement and direction. The mouse dr power from the +5V pin, while the remaining pin goes to ground.

that the packet format it employs does not include a clock bit as the Microsoft packet does. Furthermore, because the same system board controller is used to decode data coming from the mouse and the keyboard simultaneously, programs that use both the mouse and keyboard on the PS/2 sometimes experience data-overrun conditions that cause the mouse driver to lose synchronicity with the mouse. If there were a clock bit, timing could be reestablished and the overrun condition could be defeated. Without the clock bit, application programmers have been forced to resort to several ad hoc means of maintaining the integrity of the information flowing to the mouse driver. Some of these methods are more effective than others. If your mouse goes haywire or simply stops working when you run certain application programs on a PS/2 (especially older programs), it's probably for this reason.

It's no accident that you can replace the end adapter on a Microsoft Serial Mouse and plug it into the mouse port of a PS/2. The PS/2 mouse port has six pins, one of which carries power to the mouse. The Microsoft Serial Mouse looks at this line to determine what type of port it's connected to and adjusts the format of the data it sends out accordingly. If the line is asserted (carrying voltage), then the mouse knows it's attached to a PS/2 port. If the line is not asserted, the mouse concludes that it must be attached to a serial port.

THE MOUSE PROGRAMMING INTERFACE

Fortunately, programs don't have to interface with the mouse at the packet level; that's what mouse drivers are for. The mouse driver is the program you install before running an application that uses the mouse. The Microsoft mouse driver is called MOUSE.SYS or MOUSE.COM. (The only difference between the two is that MOUSE.SYS is installed with a DEVICE statement in CONFIG.SYS, while MOUSE.COM is installed from the command line or your AUTOEXEC.BAT file.) Microsoft and Microsoft-compatible mouse drivers provide a set of 35 high-level function calls that applications can use to communicate with the mouse. These functions are summarized in Figure 3.

For example, if a program wants to determine whether a mouse button is currently held down, all it has to do is make a call to function number 5, Get Button Press Information. The driver, which fields the interrupt generated when a mouse button is pressed or released, keeps track of the state of the buttons. In response to the program's request, the driver returns an integer whose value reflects the current button status. If the left button is depressed, bit 0 is set to 1; if the right button is depressed, bit 1 is set to 1. A value of 0 in either bit indicates that the corresponding button is not depressed.

Mouse driver functions are accessed through interrupt 33h. The following code sequence portrays what a typical call to the mouse driver looks like in assembly language:

```
MOV   AX,5
MOV   BX,1
INT   33H
TEST  BX,1
JZ    NOT_DEPRESSED
  .
  .
  .
```

In general, function codes are passed in register AX, and other parameters in BX, CX, and DX. This particular sequence calls function 5 to retrieve the status of the right mouse button (BX=1 for the right button, BX=0 for the left). On return, bit 1 of the BX register is tested and a branch is made if it's set to 0, indicating the button is not currently depressed.

We don't have the space here to explore each of the function calls in detail. For more information, refer to the *Microsoft Mouse Programmer's Reference* (1989, Microsoft Press), or to *PC Magazine*'s July 21, 1987, Lab Notes column ("Mouse Software: See How They Run," page 411). However, there is one function that is so useful and so widely used by pro-

MOUSE DRIVER FUNCTION CALLS

Code	Description
0	Mouse Reset and Status
1	Show Cursor
2	Hide Cursor
3	Get Button Status and Mouse Position
4	Set Mouse Cursor Position
5	Get Button Press Information
6	Get Button Release Information
7	Set Minimum and Maximum Horizontal Cursor Position
8	Set Minimum and Maximum Vertical Cursor Position
9	Set Graphics Cursor Block
10	Set Text Cursor
11	Read Mouse Motion Counters
12	Set Interrupt Subroutine Call Mask and Address
13	Light Pen Emulation Mode On
14	Light Pen Emulation Mode Off
15	Set Mickey/Pixel Ratio
16	Conditional Off
17	(Not assigned)
18	(Not assigned)
19	Set Double-Speed Threshold
20	Swap Interrupt Subroutines
21	Get Mouse Driver State Storage Requirements
22	Save Mouse Driver State
23	Restore Mouse Driver State
24	Set Alternate Subroutine Call Mask and Address
25	Get User Alternate Interrupt Address
26	Set Mouse Sensitivity
27	Get Mouse Sensitivity
28	Set Mouse Interrupt Rate
29	Set CRT Page Number
30	Get CRT Page Number
31	Disable Mouse Driver
32	Enable Mouse Driver
33	Software Reset
34	Set Language for Messages
35	Get Language Number
36	Get Driver Version, Mouse Type, and IRQ Number

e 3: Microsoft and Microsoft-compatible mouse drivers provide 35 different function calls (functions 17 and 18 are not assigned) that a program can use to communicate with the mouse. Collectively, these function calls make up the mouse programming interface.

grammers that a discussion of the mouse programming interface couldn't be considered complete without it: function 12, Set Interrupt Subroutine Call Mask and Address.

Function 12 lets a program instruct the mouse driver to interrupt it asynchronously when the mouse is moved or a button is pressed or released. The program passes the mouse driver two objects: a *call mask* and a *subroutine address*. The call mask acts as an event filter. If the program isn't interested in mouse movements or button releases, the call mask can be set up so that the driver interrupts the program only when a button press occurs. The subroutine address is a 32-bit pointer to the subroutine that the program will use to process calls from the mouse driver. By registering itself with the mouse driver this way, a program doesn't have to constantly poll the mouse driver to see what's going on. When something happens, the program is automatically notified and given a chance to act on the event.

OF MICE AND WINDOWS

When you run *Microsoft Windows*, there's an additional layer of software placed between you and the mouse: the *Windows* kernel itself. With *Windows*' message-based architecture, a *Windows* program doesn't have to make calls to the mouse driver to receive information about the mouse. Instead, *Windows* passes the program a message whenever a mouse-related event such as the press or release of a button occurs. In all, *Windows* 3.0 defines 21 different messages that a program may receive from the mouse. These messages are listed in Figure 4.

Of the 21 mouse messages, 10 pertain to the client area of the window (the inside of the window excluding the title bar, menu bar, and scroll bars; in effect, the part of the window that a program "owns"). In general, a message is transmitted to the program's client window procedure when the mouse is moved over the client area of the window and a mouse button is clicked, released, or double-clicked. *Windows* passes additional information along with the message, including the *x*- and *y*-coordinates of the mouse pointer at the time the event occurs and the state of the left, middle, and right mouse buttons and the Ctrl and Shift keys.

Nonclient-area mouse messages are passed to a window procedure when a mouse event occurs inside a program's window but outside the client area of the window. The nonclient area includes the title bar, the menu bar, the minimize and maximize boxes, the control menu box, the scroll bars, and the window's border. In general, these messages parallel the client-area mouse messages and even have similar names. However, there is one nonclient-area message that stands apart from the rest: WM_NCHITTEST (Non-Client Hit Test).

This message is passed to the program any time the mouse is moved inside the program window—inside or outside the client area—before any other mouse messages are received. It contains the *x*- and *y*-coordinates of the mouse pointer. Normally, programs simply pass this message back to *Windows*, and *Windows* uses it to generate subsequent messages.

A program written for *Windows* may choose to act on or ignore any of the mouse messages it receives. In fact, it's quite feasible (and even common) for *Windows* programs to make extensive use of the mouse without processing a single mouse message. That's because *Windows* takes care of most of the mouse processing. For example, when a menu is pulled down with the mouse, it's *Windows*, and not your application, that responds to the mouse click and displays the menu. Similarly, when the mouse pointer is positioned over a button in a dialog box and clicked, the program receives notification in the form of a message saying an on-screen button was pressed.

The way software interfaces with the mouse depends on the environment in which it's being run. All in all, the mouse is an amazing little piece of hardware, no matter where or how it's being run.

WINDOWS MOUSE MESSAGES

Client Area Mouse Messages

Message name	Sent when . . .
WM_MOUSEMOVE	The mouse is moved
WM_LBUTTONDBLCLK	The left mouse button is double-clicked
WM_LBUTTONDOWN	The left mouse button is pressed
WM_LBUTTONUP	The left mouse button is released
WM_MBUTTONDBLCLK	The middle mouse button is double-clicked
WM_MBUTTONDOWN	The middle mouse button is pressed
WM_MBUTTONUP	The middle mouse button is released
WM_RBUTTONDBLCLK	The right mouse button is double-clicked
WM_RBUTTONDOWN	The right mouse button is pressed
WM_RBUTTONUP	The right mouse button is released

Nonclient Area Mouse Messages

Message name	Sent when . . .
WM_NCMOUSEMOVE	The mouse is moved
WM_NCLBUTTONDBLCLK	The left mouse button is double-clicked
WM_NCLBUTTONDOWN	The left mouse button is pressed
WM_NCLBUTTONUP	The left mouse button is released
WM_NCMBUTTONDBLCLK	The middle mouse button is double-clicked
WM_NCMBUTTONDOWN	The middle mouse button is pressed
WM_NCMBUTTONUP	The middle mouse button is released
WM_NCRBUTTONDBLCLK	The right mouse button is double-clicked
WM_NCRBUTTONDOWN	The right mouse button is pressed
WM_NCRBUTTONUP	The right mouse button is released
WM_NCHITTEST	The mouse is moved over any part of the window

Figure 4: *Windows* 3.0 defines 21 different mouse messages that can be passed to a *Windows* application. Of the 21 messages, 10 pertain to the client window and 11 to the area outside the client window. Client-area mouse messages are received when the action takes place inside the client window; nonclient area messages are passed when the action occurs outside the client window.

ASK THE TUTOR

The Tutor solves practical problems and explains techniques for using your hardware and software more productively. Questions about DOS and systems in general are answered here. To have your question answered, write to Tutor, *PC Magazine*, One Park Avenue, New York, NY 10016, or upload it to PC MagNet (for access information, see the PC MagNet News page immediately following the Uti column). We regret that we're unable to answer questions individually. ■